Columbia's Awk Replacement Language (CARL)

Language Reference Manual

COMS 4115 Programming Languages and Translators (Spring 2017)

Darren Hakimi (dh2834)	System Architect
Keir Lauritzen (kcl2143)	Manager
Leon Song (ls3233)	Tester
Guy Yardeni (gy2241)	Language Guru

- Citations:
- "The GNU Awk User's Guide, https://www.gnu.org/software/gawk/manual/gawk.html
- Brian Kernigan and Dennis Ritchie, "The C Programming Language," Prentice Hall, 2nd Edition, 1988.

1. Introduction

This language reference manual describes Columbia's Awk Replacement Language (CARL), which, as made evident by the language name, is an implementation of the AWK language. CARL came to fruition as part of a group project for the Columbia course, Programming Languages & Translators taught by Stephen Edwards. The initial purpose behind the creation of CARL is for it to be used as a web scraper. Similar to AWK, CARL uses a `pattern {action}` model to manipulate file data. CARL can scan through files i.e. HTML files and use regular expressions to identify data. We will use OCaml to create the compiler for CARL with a target backend of LLVM.

2. Lexical Conventions

2.1 Comments

CARL only consists of single line comments. Comments are introduced by #, with no intervening blanks, and is terminated by the end of the line. Therefore, CARL ignores the remainder of the liner after the comment symbol.

2.2 Identifiers

Identifiers are a sequence of letters, digits, and underscores (_) that explicitly begin with a letter or underscore. There are 52 possible letters, being that they can be uppercase or lowercase and from the ASCII set.

2.3 Keywords

The following identifiers are keywords and are restricted from use otherwise:

`BEGIN, END, if, elseif, else, for, for..in, break, continue, return, RS, FS, NR, NF, RSTART, RLENGTH, MF, in, function`

The following character sequences are also keywords:

`{, }, [, ], (, ), =, ==, != +=, -=, *=, %=, ^=, **==, >, <, >=, <=, &&, ||, *, /, %, +, -, ^, **, ;, $, $0, $1, $2, …, ,, ‘, “, \, ~, !~, !`

2.4 Integer Constants

Integers consist of a sequence of digits, 0-9, that do not begin with 0. Integers will all be base 10. There will not be any built-in conversions to any other bases, such as octal or hex. Negative integers are prefixed by a dash (`-`).

2.5 Float Constants

Floats consist of an integer component followed by a decimal point (`.`) and a fraction component. The integer and fraction component both consist of a set of digits, 0-9. Either the integer component or the fraction component could be omitted. If the integer component is omitted, the mandatory fraction component must be preceded by the decimal point.

2.6 String Literals

A String consists of a sequence of zero or more chars and is surrounded by double quotes (“. . .”). Strings offer several escape sequences that are represented by chars within the string:

Newline (`\n`), tab (`\t`), double quote (`\”`), single quote (`\’`), backslash (`\`)

2.7 Regexp Constants

A regexp consists of expressions that are used for pattern matching. A regular expression is always introduced and terminated with a forward slash (`/`). A range pattern is defined by [X_1-X_2], such that X_1 and X_2 are defined at either uppercase letters, lowercase letters, or digits. X_1 must have a lower value than X_2 on the ASCII table. Regular Expressions can be compared using the `~` and `!~` operators to respectively determine if and if not a regexp matches another.

3. Syntax Notation

The syntax notation of this reference manual for all code, including words, characters, regular expressions and keywords, will be typed in the `Inconsolata` font. Code formatting and notation will be represented with *dark grey and italicized Times New Roman* font.

4. Meaning of Identifiers

Identifiers are names that refer to corresponding variables, functions, or actions. Refer to section 2.2 for a precise definition of identifiers.

Variable and function identifiers have inferred types, which implies that identifiers will not have type declarations and that the compiler will determine the type.

4.1 Storage Scope

The identifier storage scope is split between local and global. Local scope identifiers are declared inside of code blocks. Local identifiers can only be accessed within the block they are declared. Global scope identifiers are initialized specifically within a BEGIN or END block. Global identifiers can be accessed anywhere in the code that is below the declaration of the identifier. Functions will normally be global scope, but if the function parameter contains the variable with

The identifier storage scope is entirely global. All identifiers are initialized specifically within a BEGIN or END block. Global identifiers can be accessed anywhere in the code that is below the declaration of the identifier.

5. Objects and Lvalues

An Object is a named storage location and a lvalue is a reference to an object. In CARL, the lvalue is on the left hand side of the declaration and represented by a variable or array element. The objects types in CARL are integer, float, string, associative array, and functions.

6. Conversions

6.1. String to Integer Conversion

CARL will support string-to-integer conversion, **but not integer-to-string** conversions.

CARL converts strings to numbers by interpreting any numeric prefix of the string as numerals e.g. str = “8371this is an example” will convert str to the integer 8371. CARL will also support scientific notation conversions as well e.g. the string “1e3” converts to the integer 1000.

6.2. Arithmetic Conversion

Only two numeric data types are supported: int and float. For arithmetic operations, if one operand is a float, the other operand will be converted to a float as well

7. Rule

Like Awk, CARL programs consists of a series of rules. A rule specifies one pattern to look for, and one action to perform when the pattern is found. Actions are enclosed in braces to separate it from the pattern definition. Rules are separated by newlines. Thus, a series of rules will look like this:

rule:
pattern { action }
pattern { action }
pattern { action }

CARL reads the input file one line at a time. For each line, CARL looks for the patterns in each rule, trying patterns in the order that they were written. If no pattern matches, then no action is taken. Once CARL attempts searching for every pattern, it moves on to the next line.

8. Patterns

Patterns control the execution of actions in a rule -- CARL performs an action on an input record every time it matches a pattern. CARL supports similar types of patterns to awk. These are:

1. Regex Patterns
2. Expression Patterns
3. Range Patterns
4. Special BEGIN/END Patterns
5. Empty Pattern

8.1 Regex Patterns

A Regex pattern simply contains a regex constant in the pattern definition of a rule. Remember that a rule just looks like this: *Pattern { Action }*

So a rule with a regex pattern might look like this:

```
/[A-Z]/ { print “wow!” }
```

This rule prints “wow!” for any record that contains a capital letter.

8.2 Expression Patterns

CARL expressions are valid as patterns. An expression is a match if it’s value evaluates to non-zero (if it is a number) or non-null (if it is a string). The expression is reevaluated every time CARL tests a new record. Expressions can use variables or fields from the input record (e.g. $3).

Comparison expressions are a common pattern. Consider the following:

```
$2 == /[A-Z]/ { print “wow! }
```

This rule only looks for capital letters in the second field of the record, and prints “wow!” if the pattern matches.

Boolean expressions such as this one are very useful as well:

```
/columbia/ && /.edu/ { print “student from columbia” }
```

This rule makes sure the record contains both the words “columbia” and “edu” before executing the action.

8.3 Range patterns

A range pattern is defined by two patterns, separated by a comma.

```
beginpat , endpat { Action}
```

We use range patterns when we want to match a range of consecutive records. When a record matches *beginpat*, the range pattern is turned on, and CARL performs the action for every consecutive record. CARL continues scanning records until it finds a record that matches *endpat*, then it turns the range pattern off.

Note that matching for range patterns is *inclusive*, meaning that the records that match *beginpat* and *endpat* will execute the action.

You can use range patterns alongside other rules as well. Consider the following:

```
/[A - Z]/ { print “capital letter found” }

/hello/ , /goodbye/ { print “range is ON” }

/[0-9]/ { print “numeric character found” }
```

This is a series of three rules. Two of the rules use simple regex patterns, while the middle rule uses a range pattern.

CARL will try to match these three patterns for every record. Once it finds a record with “hello”, it turns the range pattern on, and prints “range is ON” for every record until it comes across another record with “goodbye”. As CARL is doing this, it continues to match patterns from the other rules as well i.e. look for any capital letters or numbers in every subsequent record while still searching for “goodbye”. Note that only after “goodbye” is found does CARL try to search for “hello” again.

It is also possible for a range pattern to be turned on and off by the **same** record, in which case the action is executed for that record only.

8.4 BEGIN/END

BEGIN and END are keywords reserved to define special patterns. The BEGIN rule is executed once before any input is read, and similarly the END rule is executed after every input is read. Think of them as the startup and cleanup actions for a CARL program.

CARL allows for multiple BEGIN and END rules, running them in the order which they are written. Note that BEGIN and END cannot be used with any operators or expressions.

8.4.1 I/O issues when using BEGIN/END

Remember that BEGIN actions run before any input is read, so variables like $0 are undefined because there are no input records, and hence no fields yet.

8.5 The Empty Pattern

The empty pattern matches **every** input record. You can define it by declaring an action without a corresponding pattern e.g. simply do:

{ print $1 }

Which prints the first field in every record.

9. Expressions

9.1. Primary Expressions

Primary expression are the same as in C: identifiers, constants, strings, and (expressions).

primary-expressions:
identifier
constant
(expression)

Identifiers are primary expressions and defined by the rules in Section 2.2. Identifiers are not declared, rather they are created when initialized using the assignment operator. Constants are primary expressions. They are defined in Section 2. Strings are also primary expression and are enclosed double quotes "`string`". Parenthesized expressions are primary expressions.

Regex constants are also primary expressions. Regex

9.2. Prefix Field Reference

When CARL processes a file and record matches the pattern, they the record is further parsed into fields based on a field separator.

prefix-field-expression:
primary-expression
$prefix-field-expression

The field separator is nominally whitespace, but can be assigned at run time. The fields can then be addressed positionally using a field reference. Field references are ones-indexed and $0 is reserved for the entire record. Field reference must be numeric expressions.

9.3. Postfix Expressions

Postfix expressions are as follows and all operators are left associative:

postfix-expression:
prefix-field-expression
postfix-expression[expression]
postfix-expression(argument-expression-list)
postfix-expression++
postfix-expression--
argument-expression-list:
assignment-expression
argument-expression-list, assignment-expression

9.3.1. Array References

Array references can be any expression, but numeric expressions will be converted to strings prior to hashing in the associative array.

9.3.2. Function Calls

Function calls are the same as C. Multiple assignment-expressions can be passed to functions. All expressions are fully evaluated (assignment-expressions are at the bottom of the context-free grammar) before passing to the function. No spaces may be between postfix-expression and left parentheses. Arguments are separated by commas.

Arguments are fully evaluated and passed by value to the function call. Functions must be defined prior to being called. The types for function called will be inferred based on the operations of the function as defined in Section 10. Functions only have the scope of the function code.

9.3.3. Postfix Incrementation

Postfix incrementation is the same as C. With a postfix incrementor, the value is provided then incremented. This precedence differs slightly from GAWK.

9.4. Prefix Incrementation

Prefix incrementor operators function the same as C. The value is incremented by one and then returned.

prefix-inc-expression:
postfix-expression
++prefix-inc-expression
--prefix-inc-expression

9.5. Exponentiation Operators

CARL supports exponentiation using both ** and ^, although ^ is recommended. Exponentiation operators group right to left.

exponentiation-expression:
prefix-inc-expression
*prefix-inc-expression ** prefix-inc-expression*
prefix-inc-expression ^ prefix-inc-expression

Exponentiation requires both operands to be numerics.

9.6. Unary Operators

Unary operators are right associative and are as follows:

unary-expression:
exponentiation-expression
unary-operator unary- expression
unary-operator: one of
+ - !

Unary-expressions must be numeric for all operators.

9.6.1. Unary Plus Operator

This does nothing but it included for symmetry with minus operator.

9.6.2. Unary Minus Operator

Changes the sign of the unary-expression. Operand must be numeric.

9.6.3. Logical Negation Operator (!)

CARL does not have Boolean values, like C any value greater than zero is considered true and only exactly zero is considered false. Logical negation converts all values of 0 to 1 and all values > 0 to 0. Empty strings are considered false while all other strings are considered true.

9.7. Multiplicative Operators

Multiplicative operators associate left to right and include *, /, and %. Division results in integer to floating point conversion unless both operands are constants and the result is an integer.

Modulus automatically results in an integer. Multiplication of two integers results in an integer, while multiplication of one or more floating points results in a floating point.

multiplicative-expression:
unary-expression
*multiplicative-expression * unary-expression*
multiplicative-expression / unary-expression
multiplicative-expression % unary-expression

Expressions must be numeric.

9.8. Additive Operators

Additive operators associate left or right and include + and -.

additive-expression:
multiplicative-expression
additive-expression + multiplicative-expression
additive-expression - multiplicative-expression

9.9. String Concatenation

String concatenation occurs when two strings are separated by a ' '. The string is formed from right to left.

string-concat-expression:
additive-expression
string-concat-expression string-concat-expression

Both expressions must be strings or numerics with implicit conversion to strings. Strings are concatenated into a single string.

9.10. Relational and Equality Operators

The standard relational operators <, <=, ==, =>, >, != are available in CARL. Relational operators are evaluated left to right.

relational-expression:
string-concat-expression
relational-expression relational-operator string-concat-expression
relational-operator: one of
<, <=, ==, =>, >, !=

The relational operators apply to both numeric values and string. The operators return 1 if true and 0 if false. Strings are compared in lexicographical order character by character. Comparison between floating-point numbers and integers results in the conversion of the integer to a floating-point number.

The greater-than symbol (>) is also used for redirection for print statements. The context is determined automatically.

9.11. Matching Operators

Matching operators are used to compare strings to regexp constants. If the regular expression matches the string then the expression returns a 1, otherwise it returns a 0.

matching-expression:
relational-expression
matching-expression ~ string-concat-expression
matching-expression !~ string-concat-expression

One of the operands must be a regexp constant while the other must be a string. Operators are left associative.

9.12. Array membership

Associative arrays member is tested using the keyword in: `key in array`, where key is a potential key and array is an associative array. The operator is evaluated left to right.

array-member-expression:
matching-expression
string-concat-expression in primary-expression

Array names cannot be built from strings and numerics for the purpose of membership tests. The array name must be explicitly defined.

9.13. Logical AND Operator (&&)

The logical AND operator uses the symbol &&. Logical AND returns true, if and only if both expressions are true.

logical-and-expression:
array-member-expression
logical-and-expression && array-member-expression

The left-side expression is evaluated first. If it is 0, then the right-side expression is never evaluated and the expression returns 0 (the operator is said to be short circuited.) If any side effect behavior from the left side would occur (for example, ++), then this side effect does not occur. If the left-side expression is true (numeric greater than 0, non-empty string), then the right-side expression is evaluated. The are no bitwise operators as in C. The operator is evaluated from left to right.

9.14. Logical OR Operator (||)

The logical OR operator uses ||. The logical OR returns true if either expression is true.

logical-or-expression:

logical-and-expression

logical-or-expression || logical-and-expression

Logical OR operators are short-circuited in the same was as logical AND operators, except that if the left-side operator is true, then it automatically returns 1. Operations are evaluated from left to right.

9.15. Conditional Operator

CARL supports conditional operators (short form if statements). These function the same as C.

conditional-expression:

logical-or-expression

logical-or-expression ? expression : conditional-expression

The logical-or-expression is evaluated and then either expression or conditional expression is chosen. Like GAWK, these group right to left.

9.16. Assignment Expressions

Assignments must be to prefix-inc-expressions. Operations are evaluated from right to left, so a = b = 4 is equivalent to a=(b=4).

expression:

conditional-expression

prefix-inc-expression assignment-operator expression

assignment-operator: one of

=, +=, -=, *=, %=, ^=, **=

Assignment operators complete the operation in the operator using the value of the left-side and the value of the right side then storing it back in the left hand operator. "/=" has been removed to eliminate the ambiguity with regular expressions. Assignment operators return their assigned value.

9.17. Print Operator

The `print` and `printf` functions in CARL do not use parentheses, unlike C but like GAWK. This design decision was due to the common use of printing statements. They can print multiple expressions that are separated by commas. Expressions are evaluated left to right and printed in order.

print-expression:

expression

print print-expression-list

printf string-concat-expression, print-expression-list

print-expression-list:

expression

print-expression-list, expression

`Print` expressions contain only comma-separated expressions and each is printed in turn. The first argument after `printf` must be a string.

10. Declarations

A declaration introduces new entities into the program and identifies them. The type of the entity is inferred based on the value of the entity upon assignment.

The following entities are available for declaration:

10.1 Data types:

Integers:

An integer type will be inferred by having a value within (-)[0-9]+ with the possibility of being negative.

Example: my_int = 5

Floating points:

A floating point type will be inferred by containing a decimal within (-)[0-9]*.[0-9]* with the possibility of being negative.

Example: my_float = 4.123

Strings:

A string type will be inferred by the opening and closing of two quotation or two apostrophe characters.

Example: my_language = “CARL”

Associative arrays:

An associative array type is inferred through curly brackets. An associative array can either be one with integers for indices or with strings for indices.

Example: my_array = {1 : “C”, 2 : “A”, 3 : “R”, 4 : “L”}

10.2 Functions:

Functions:

A Function type is inferred by being wrapped in parenthesis and curly brackets wrapping the function's contents.

```
Example: function my_func(value1, value2)

{

new_value = value1 + value2;

return new_value;

}

My_variable = myfunc(arguments);
```

11. Statements

Statements are executed in order from top to bottom, left to right.

11.1 Assignment Statements:

Assignment statements specify an identifier and its equality, ending in semicolon.

```
Identifier = value;
```

11.2 Conditional Statements:

Conditional statements evaluate expressions and execute other statements based on if the expression is true or false.

```
if ( expression ) { statement }

if ( expression ) { statement } else {statement}

if ( expression ) { statement } elseif ( expression ) {statement} else {statement}
```

11.3 For Statements:

For statements execute the specified body statement based on a condition. "expression1" is used for initialization. "expression2" is used as a condition, "expression3" is used as an

increment. The body statement will continue re-executing as long as the condition set in expression2 is true.

```
for ( expression1 ;  expression2 ; expression3 ) { statement }
```

11.4 Break Statements:

A Break statement will terminate the enclosing For statement. It is usually used in a For statement when a certain condition occurs.

```
break;
```

11.5 Continue Statements:

A continue statement will jump to the next cycle of a For statement, skipping the rest of the body of the For statement below it. It is usually used in a For statement when a certain condition occurs.

```
continue;
```

11.6 Return Statements:

A return statement will return from a function. A return statement can either return nothing or return a value from an expression.

```
return;

return {expression};
```

12. Scope

12.1 Variable Scope In Functions:

Global:

Variable scope for functions in CARL is global.Initialized variables used in block statements or functions and modified within will have their value changed on a global level.

Example:

```
function my_func() {
        i = 6;
        print "i func is equal to: "i;
}
i = 5;
my_func();
print "i global is equal to: "i;
```

Results:

```
i func is equal to: 6
i global is equal to: 6
```

Local:

In order to keep the value of a variable unchanged on the global level when it is used in a function, it can be passed as an argument to the function and prepended with with a pipe character.

Example:

```
function my_func(|i) {
        i = 6;
```

```
                print "i func is equal to: "i;
        }
        i = 5;
        my_func();
        print "i global is equal to: "i;
Results:
        i func is equal to: 6
        i global is equal to: 5
```

12.2 Variable Scope In Action:

Variables declared within the BEGIN and END sections are global, while variables declared in between them are local.

13. Grammar

Basic-unit:
- *Declaration*
- *Rule*

Declaration:
- *function-definition*
- *variable-definition*

Function-definition:
- *declaration*
- *declaration-list*
- *statement*
- *parameter-list*

Rule:
- *Pattern - { Action }*

Pattern:
- *BEGIN/END*
- *Empty Pattern*
- *Regex Pattern*
- *Expression Pattern*
- *Range Pattern*

Action:

Statement
Function-call

Statement:
print-expression
Assignment Statement
Conditional Statement
For statement
Return statements
Break Statements
Continue Statements
print-expression:
expression
print print-expression-list
printf string-concat-expression, print-expression-list
print-expression-list:
expression
print-expression-list, expression
expression:
conditional-expression
prefix-inc-expression assignment-operator expression
assignment-operator: one of
*=, +=, -=, *=, %=, ^=, **=*
conditional-expression:
logical-or-expression
logical-or-expression ? expression : conditional-expression
logical-or-expression:
logical-and-expression
logical-or-expression || logical-and-expression
Logical-and-expression:
array-member-expression
logical-and-expression && array-member-expression
array-member-expression:
matching-expression
string-concat-expression in primary-expression
matching-expression:
relational-expression
matching-expression ~ string-concat-expression
matching-expression !~ string-concat-expression
relational-expression:
string-concat-expression
relational-expression relational-operator string-concat-expression
relational-operator: one of

<, <=, ==, =>, >, !=

string-concat-expression:

additive-expression

string-concat-expression string-concat-expression

Additive-expression:

multiplicative-expression

additive-expression + multiplicative-expression

additive-expression - multiplicative-expression

Multiplicative-expression:

Unary-expression

*multiplicative-expression * unary-expression*

multiplicative-expression / unary-expression

multiplicative-expression % unary-expression

unary-expression:

exponentiation-expression

unary-operator unary- expression

unary-operator: one of

+ - !

prefix-inc-expression:

postfix-expression

++prefix-inc-expression

--prefix-inc-expression

postfix-expression:

prefix-field-expression

postfix-expression[expression]

postfix-expression(argument-expression-list)

postfix-expression++

postfix-expression--

primary-expressions:

identifier

(expression)

argument-expression-list:

expression

argument-expression-list, expression

Identifier:

Constant

Function-definition

Constant:

integer-constant

float-constant

Regexp-constant

string constant

14. Examples

15.1 Email parsing example:

```
carl '$0, /.*@.*\..*/ { print MF}'
```

replaces the longer

```
awk 'match($0, /.*@.*\..*/) { print substr( $0, RSTART, RLENGTH )}'
```

15.2 Phone number parsing example:

```
carl'$0, /\([0-9]{3}\)[[:space:]][0-9]{3}\-[0-9]{4}/ { print MF}'
filename
```

replaces the longer awk version:

```
awk 'match($0, /\([0-9]{3}\)[[:space:]][0-9]{3}\-[0-9]{4}/) { print
substr( $0, RSTART, RLENGTH )}'
```

15.3 Text database parsing example:

Input file:

```
1.     (212)-123-1011     $120,000    Andrew       M
2.     (222)-298-8494     $50,000     Bob          M
3.     (114)-124-1234     $170,000    Katy         F
4.     (123)-222-1111     $90,000     Lisa         F
5.     (123)-132-4666     $80,000     Billy        M
6.     (222)-115-1515     $130,000    Alexa        F
7.     (718)-123-4556     $242,000    Mia          F
```

Code:

```
BEGIN {
       total=0;
        highEarners=0;
       lowEarners=0;
       females=0;
       males=0;
}
{
       gsub(/\$|,/,"",$3);
       gsub(/\(|\)|-/,"",$2);
       personID=$1;
       phoneNum=$2;
       salary=$3;
       name=$4;
       gender=$5;
       total=total+salary;
       if (salary>=100000) {
              highEarners=highEarners+1;
       } else {
              lowEarners=lowEarners+1;
       }
       if (gender=="F") {
              females=females+1;
       } elseif (gender=="M") {
              males=males+1;
       }
       areaCode = substr(phoneNum, 0, 3);
       if (dict[areaCode]) {
              dict[areaCode] = dict[areaCode]+1;
       } else {
              dict[areaCode] = 1;
       }
}
END    {
```

```
        print "Total Amount: $"total;
        average=total/NR;
        print "Average Amount: $"average;
        print "High Earners: "highEarners;
        print "Low Earners: "lowEarners;
        print "Male to Female Ratio: "males":"females;
        for (i in dict) {
              print "Area code: "i" has "dict[i]" people.";
        }
}
```

Output (by running by "carl -f [code-file] [data-file]":

```
Total Amount: $882000
Average Amount: $126000
High Earners: 4
Low Earners: 3
Male to Female Ratio: 3:4
Area code: 222 has 2 people.
Area code: 114 has 1 people.
Area code: 212 has 1 people.
Area code: 718 has 1 people.
Area code: 123 has 2 people.
```

Notes:

Changes from GAWK:

1. Removing the /= arithmetic assignment operator, which is ambiguous with REGEX.
2. C precedence is used for incrementors, where postfix has a higher precedence than prefix. In GAWK they are equal precedence.
3. All numbers in GAWK are floating point numbers. CARL will have Integer type and floating point type and the type will be infered
4. No bitwise comparison operators | and |&
5. CARL will allow the usage of elseif in an if-else statement.
6. CARL will allow only For loop statements.
7. CARL will allow the construct
8. CARL allows variables to be local to functions without them being changed globally by prepending the variable argument with | whereas AWK would conventionally prepend the variable argument with an extra space.